**Practical No. 7**

**Aim:** To implement Producer Consumer problem in Java

**Theory:**

In [computing](https://en.wikipedia.org/wiki/Computing), the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-[process](https://en.wikipedia.org/wiki/Process_(computing)) [synchronization](https://en.wikipedia.org/wiki/Synchronization_(computer_science)) problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size [buffer](https://en.wikipedia.org/wiki/Buffer_(computer_science)) used as a [queue](https://en.wikipedia.org/wiki/Queue_(data_structure)).

* The producer’s job is to generate data, put it into the buffer, and start again.
* At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of [inter-process communication](https://en.wikipedia.org/wiki/Inter-process_communication), typically using [semaphores](https://en.wikipedia.org/wiki/Semaphore_(programming)). An inadequate solution could result in a [deadlock](https://en.wikipedia.org/wiki/Deadlock) where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers

**Program:**

import java.util.LinkedList;

public class Threadexample

{

public static void main(String[] args)

throws InterruptedException

{

final PC pc = new PC();

// Create producer thread

Thread t1 = new Thread(new Runnable()

{

@Override

public void run()

{

try

{

pc.produce();

}

catch(InterruptedException e)

{

e.printStackTrace();

}

}

});

// Create consumer thread

Thread t2 = new Thread(new Runnable()

{

@Override

public void run()

{

try

{

pc.consume();

}

catch(InterruptedException e)

{

e.printStackTrace();

}

}

});

// Start both threads

t1.start();

t2.start();

// t1 finishes before t2

t1.join();

t2.join();

}

public static class PC

{

// Create a list shared by producer and consumer

// Size of list is 2.

LinkedList<Integer> list = new LinkedList<>();

int capacity = 2;

// Function called by producer thread

public void produce() throws InterruptedException

{

int value = 0;

while (true)

{

synchronized (this)

{

// producer thread waits while list

// is full

while (list.size()==capacity)

wait();

System.out.println("Producer produced-"

+ value);

// to insert the jobs in the list

list.add(value++);

// notifies the consumer thread that

// now it can start consuming

notify();

// makes the working of program easier

// to understand

Thread.sleep(1000);

}

}

}

// Function called by consumer thread

public void consume() throws InterruptedException

{

while (true)

{

synchronized (this)

{

// consumer thread waits while list

// is empty

while (list.size()==0)

wait();

//to retrive the ifrst job in the list

int val = list.removeFirst();

System.out.println("Consumer consumed-"+ val);

notify();

Thread.sleep(1000);

}

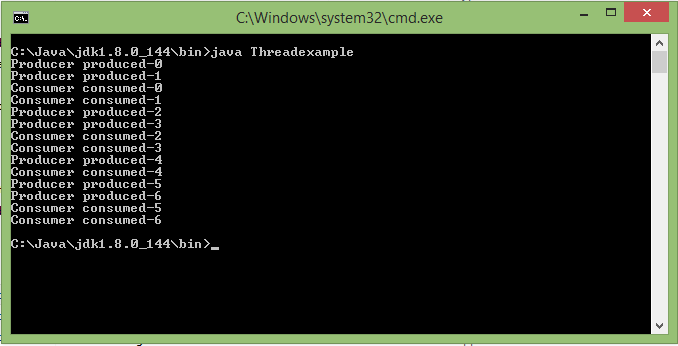
}

}

}

}

**Output:**

****

**Conclusion:** The producer consumer problem has been implemented using threads in Java.